5-1-2012

Android Application for Cluster Job Management

Christopher Schance
Boise State University

ANDROID APPLICATION FOR CLUSTER JOB MANAGEMENT

by

Christopher Schance

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2012

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the project submitted by

Christopher Schance

Project Title: Android Application for Cluster Job Management

Date of Final Oral Examination: 4 May 2012

The following individuals read and discussed the project submitted by student Christopher Schance, and they evaluated their presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Teresa Cole, Ph.D.	Member, Supervisory Committee
Jim Buffenbarger, Ph.D.	Member, Supervisory Committee

The final reading approval of the project was granted by Amit Jain, Ph.D., Chair, Supervisory Committee. The project was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

I would like to thank Amit Jain, for his guidance during my graduate college career and allowing me the opportunity to pursue this project. Thanks also goes to Teresa Cole and Jim Buffenbarger for agreeing to serve as committee members. All three contributed to helping me obtain skills which facilitated the completion of this project.

Special thanks goes to my wife, Starrita, for her understanding and support throughout all my entire college experience. Without her, this would not have been possible.

Finally, I would like to thank my mother, Judith, who introduced me to computer programming when I was in fifth grade, by teaching me the BASIC programming language on a TRS-80 computer. Subsequently, she also helped me purchase my first computer, an Atari 800XL, which I enjoyed programming and playing games with for many years through junior high and high school.

ABSTRACT

Compute clusters are used to solve large, computation-intensive problems. These systems are shared due to a large investment in resources to set up and maintain these systems. Often, a job queue or batch system, such as a Portable Batch System (PBS) is used to enable sharing the cluster resources among many researchers. This project provides a mobile application which allows the user to view the status of the jobs on a cluster. Also, the ability to perform some simple administrative tasks is included. The application is written for Android devices, using the Java programming language. This is a good case study for developing a medium complexity application with performance issues.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADT – Android Development Toolkit

CLI – Command Line Interface

CPU – Central Processing Unit

DDT – Design Driven Testing

GUI – Graphical User Interface

IDE – Integrated Development Environment

MPI – Message Passing Interface

ORM – Object Relational Mapper

PBS – Portable Batch System

PVM – Parallel Virtual Machine

SDK – Software Development Kit

SQL – Structured Query Language

SSH – Secure Shell

TDD – Test Driven Development

UI – User Interface

XML – Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Background

The proliferation of mobile devices enabled a level of convenience which has not been experienced before. Many applications produced for these devices have provided ways for people to utilize their available time more effectively. Simple tasks can now be completed during moments that were not usable before, while riding a bus, in an elevator, or at a restaurant waiting for your party to arrive.

For those who manage computer clusters, a mobile application would make it possible to monitor the cluster while not in the office or in front of a computer terminal. Such an application could provide simple functionality that would allow one to do simple job management tasks and be aware of how the cluster is functioning, while away from the office or lab.

This project produces such an application. The following sections in this chapter provide background information to set the stage for understanding the details of the project design, implementation, challenges, and lessons learned.

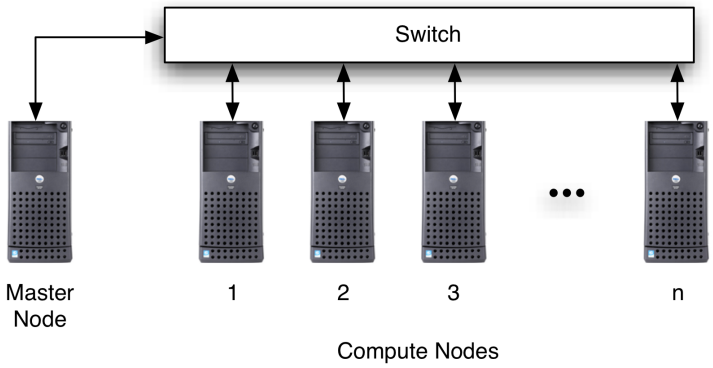

Figure 1.1: A Beowulf Cluster Configuration

1.2 Computer Clusters

A computer cluster is composed of an interconnected network of workstations or personal computers, which are then used to solve computation-intensive problems. One architecture for a computer cluster is the Beowulf architecture, which is illustrated in Figure 1.1. The Beowulf architecture has a *master node* and a number of *compute nodes*. The software written to run on these clusters usually employs message-passing software such as Message Passing Interface (MPI) or Parallel Virtual Machine (PVM) [16].

There is cost associated with building and maintaining a cluster. For that reason, clusters are often shared among many researchers and compute jobs are placed in a batch system. Often, a job queue or batch system, such as a Portable Batch System (PBS) is used to enable sharing the cluster.

1.3 Mobile Devices

Developing applications for mobile devices comes with a unique set of challenges. Understanding how the application is expected to behave in this environment is

paramount to building a useful and successful application. The remaining sections of this chapter lay the groundwork for understanding how these challenges may be addressed.

1.4 Activity Lifecycle

For the vast majority of Android applications, there is a graphical user interface which gets displayed on the screen of the device. An *Activity* represents all graphical components that are displayed on the screen at any given moment. As a user engages with the application, many *Activities* may be traversed.

Since the user can switch between applications in a fairly unpredictable manner, understanding the life-cycle an *Activity* can go through is important. Fortunately, Android provides a framework to handle this behavior. Figure 1.2 depicts a simplified version of the Android activity life-cycle. When the life-cycle begins, the Android framework will call the *onCreate* method for the *Activity*. While at the very end of the life, the *onDestroy* method is called. Only while the *Activity* is in the *running* state, will it be in the foreground and visible on the screen. Depending on user activity and resources, the *Activity* may move to the background (paused) or even be destroyed by the OS. One thing is certain, every time the *Activity* enters the *running* state, the *onResume* method is called. And, every time the *Activity* leaves the *running* state, the *onPause* method is called by the framework.

The *Activity* class provided by the Android framework provides an interface, part of which is composed of methods listed in the *activity life-cycle*. For example, the *onCreate()* method is often overridden in subclasses of *Activity*. Understanding this diagram is very important for developing useful Android applications.

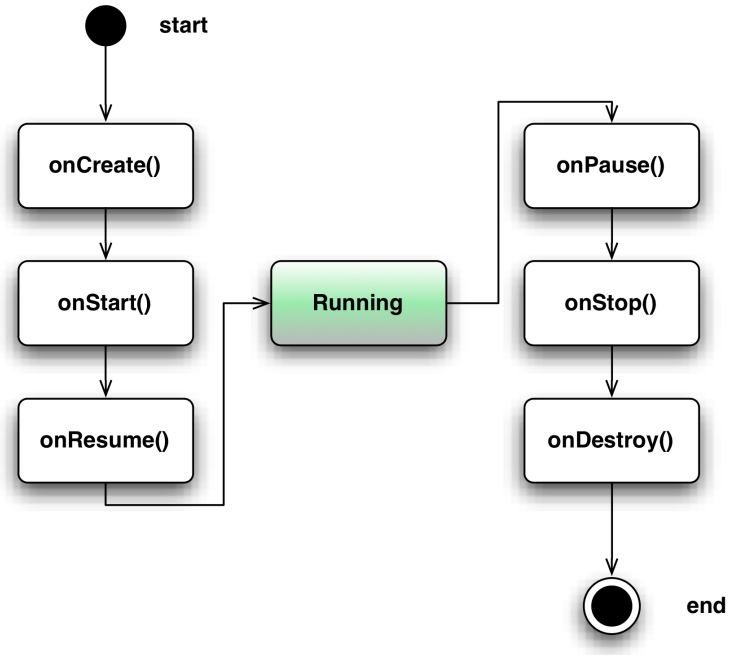

Figure 1.2: Simplified Android Activity Lifecycle

1.5 Activity Stack

When an application is started in Android, an Activity stack is associated with it. The stack represents the current *task* being performed. As new activities are started they are pushed onto the stack. The activity at the top of the stack is the activity the user sees on the screen. If the *back* button is pressed, an activity is popped from the stack, which results in the next activity being displayed on the screen.

1.6 Multitasking in Android

A *task* represents something the user is trying to accomplish and is represented by a group of *Activities*. A task may be as simple as reading email or using the web browser. Multitasking is achieved with Android when the *home* button is pressed while using

an application. This puts the current task into the background and displays the *home* screen. The user can then choose to begin another task by selecting an icon for any application. Once the icon is selected, the chosen application launches. If the newly launched application had a task in the background, the result is that the task is moved to the foreground and is now the currently active task. At most, only one task can be in the foreground. Understanding how this process works is important for application development. Activities which are part of the stack for a task that is in the background have not been destroyed and may be resumed when the application is brought to the foreground. However, the developer cannot assume the activities will not be destroyed since the operating system may destroy these activities to free up resources for other applications.

1.7 Project Statement

This project provides a mobile application for monitoring the status of jobs on a cluster and also perform simple job management tasks. The application runs on Android phones, using operating system version 2.2 or later.

1.8 Prior Work

A search on any mobile application web site will find several Secure Shell (SSH) client applications [4]. However, at the time of this writing, there doesn't seem to be any mobile applications specifically designed for monitoring and managing PBS jobs. Although an SSH client application for a mobile device could be used to monitor PBS jobs on a cluster, it would not take advantage of the typical user experience expected

from mobile applications. The user would have to use the keyboard available on the device.

Figure 1.3 shows how the UI of an SSH client application may look. Just as if the user was using their desktop computer, they are expected to type in the entire command using the CLI. Although this is flexible and allows the user to execute any command they wish, it is cumbersome on a mobile device. For example, it would take over 40 touches to delete three cluster jobs using the CLI.

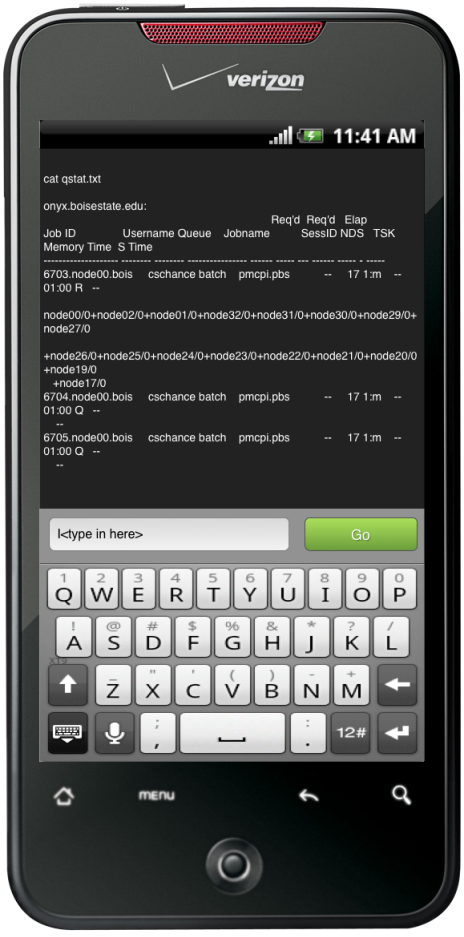

Figure 1.3: Screenshot of a Typical SSH Client Application UI

Figure 1.4 displays a screenshot of how the PBS Job Watcher application was

envisioned. This is not a screenshot of the actual application, but the final product does look similar. It has a more graphical look, compared to the UI for the SSH client app, and is more typical of mobile applications. Although, the user cannot type in any command they wish, they can accomplish most common commands with a minimal amount of interaction. For example, to accomplish the same task of deleting three cluster jobs discussed previously, the user only needs to execute about four touches.

For these reasons, an application specifically designed for monitoring and maintaining PBS jobs on a cluster would be preferred over a generic SSH client application.

Figure 1.4: Prototype of the UI for this Application

CHAPTER 2

REQUIREMENTS

2.1 Target Device

The target device for this application is an Android phone running SDK version 2.3.4. The earliest SDK for this application is version 2.0. Although this application will run on Android tablets, the user interface was not optimized for the larger screen size that a tablet device offers.

2.2 Application Behavior

This application will allow the user to view jobs running on any number of remote clusters, given they are able to establish SSH communication with a single gateway machine. An Internet connection is required for this to be possible. Once information on cluster jobs is obtained, the user can browse through the jobs and view detailed information. Also, simple operations may be performed on selected jobs.

2.3 Performance

As with any mobile application, its affect on battery usage and resources such as memory is important. The design of this application attempts to minimize the impact.

2.4 Design Patterns

Object-oriented design patterns should be used in the design, where appropriate. Using them enables flexibility, software reuse, and ease of maintenance. This application uses design patterns.

CHAPTER 3

ANALYSIS

3.1 Operating Environment

The operating environment for this application involves the mobile device on which the application is installed, the Internet, a gateway machine, and a number of clusters connected to the gateway. Figure 3.1 shows how the different entities interact.

There are a number of clusters, with each master node maintaining a queue of jobs. In PBS, the CLI provides the command *qstat* to view job information. Some items included in the job information are *job identification*, *job name*, *job owner*, and more.

Each master node can be accessed through a *gateway* machine. The *gateway* has a connection to the Internet. For users to access the gateway from the Internet, they must be approved for access to the *gateway* through SSH.

3.2 Secure Shell

Since the application will be using SSH to establish communication with the *gateway* machine, it is reasonable to explore the options for providing this ability through software. One obvious choice would be to implement the SSH2 protocol from scratch.

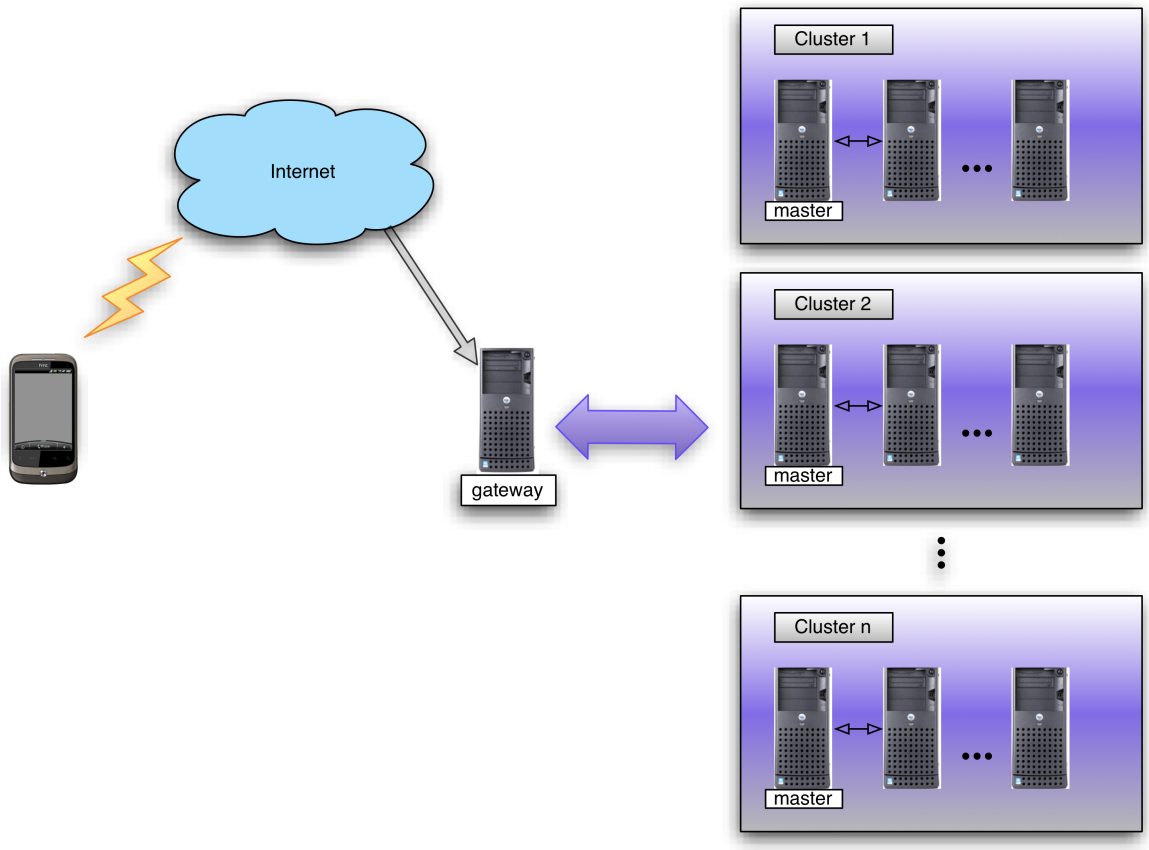

Figure 3.1: Operating Environment for the Application

Since Android programming is done using the Java programming language, it would be implemented in Java.

Another option would be to explore existing third-party packages available. This is commonplace in software development, since there are many libraries that have already been developed for reuse. Two third-party packages or libraries available are libssh2 and Java Secure Channel.

Libssh2 is a C implementation of the SSH2 protocol [7]. Since it is written in C, this library can be used with almost any programming language. Java Secure Channel is a pure Java implementation of the SSH2 protocol [6]. Since most Android

programming is done with the Java programming language, Java Secure Channel is used in this application.

3.3 Development Environment

This project was developed using an Apple MacBook Pro (13 inch, early 2011), using Java SDK version 1.6.0_29, and git version 1.7.5.4. GitHub [3] was used for a remote repository, and the GitBox (version 1.4) desktop application [12] was used for pushing to the remote repository.

The Eclipse IDE [2] was used for implementing and debugging the software. Here are the specifications for the Eclipse install used for this project.

- Eclipse IDE for Java Developers
  - Indigo Service Release 1
  - Build id: 20110916-0149
  - Android Development Toolkit (ADT)
    - The Android Open Source Project
    - 15.0.1.v201111031820-219398
    - com.android.ide.eclipse.adt
    - Dalvik Debug Monitor Service (DDMS)
    - TraceView
    - Hierarchy Viewer

CHAPTER 4

DESIGN AND IMPLEMENTATION

The first section, *Application Design*, describes a high-level view of how the application is organized. The sections that follow go into more detail on the individual components of the application design and implementation.

4.1 Application Design

Figure 4.1 shows the overall design of the application. Starting from the bottom layer, the Java Secure Channel is the third-party package which provides the implementation of the SSH2 protocol. Alongside is the SQLite database, which is a self-contained, serverless SQL database engine [10].

The second layer from the bottom contains the lowest layer that is implemented in this project. The *Cluster Communication* is supported by a set of Java classes which provide an interface to the Java Secure Channel. The *DB Adapter* provides an interface to SQLite.

Services do the heavy lifting for the application. They coordinate all of the needed network communication by using the *Cluster Communication* and the access to the data via the *DB Adapter*. The *Services* are responsible for using background threads for performing long running tasks, which improves the responsiveness of the

application. The classes in this layer extend the *Service* class provided by the Android framework, in the *android.app* package.

Finally, the *Android UI* layer is at the top. This layer provides the user interface components which are displayed on the screen of the Android device. All of the classes in this layer are subclasses of the *Activity* class provided by Android framework, in the *android.app* package. The remaining sections discuss these components in more detail, with the exception of Java Secure Shell and SQLite, which are third-party packages.

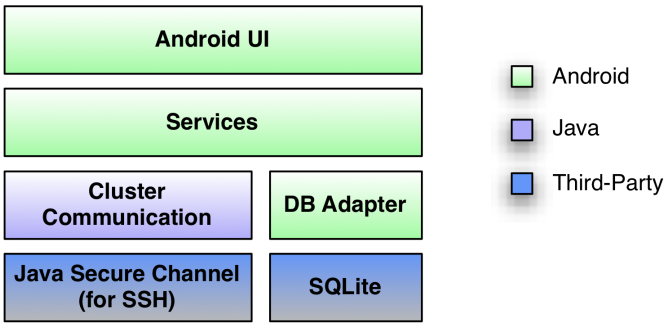

Figure 4.1: Application Design

4.2 Cluster Communication

4.2.1 Java Secure Channel

The *Java Secure Channel* package provides the classes needed to achieve SSH communication [6]. The classes described in this section provide an interface to the *Java Secure Channel* package. There are many reasons for having such an interface. For example, the interface isolates the third-party package from the rest of the application. This makes it easier to replace the *Java Secure Channel* package with a different

package that provides the same functionality. This may be desirable if another package becomes available that gives a performance advantage.

4.2.2 Adapters for Session and Channel

The *Session* and *Channel* classes are adapters to the Session and Channel mechanisms [11] which are implemented by the Java Secure Channel package.

4.2.3 Host Classes

Three classes represent the different hosts which the application will be connecting to with SSH. These classes are shown in Figure 4.2. This group of classes implements the *Composite* pattern [14]. *Gateway* plays the *Composite* role of the pattern, while *Cluster* plays the *Leaf* role. *Host* implements the default behavoir for the class hierarchy. Figure 4.3 shows the class diagram for the *Host* class.

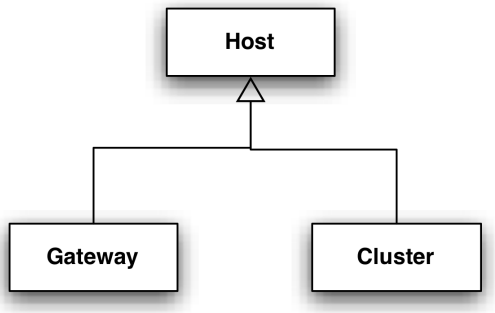

Figure 4.2: *Host* Classes

The *Gateway* class implements methods which manage the children or *Cluster* objects. The implemented methods are shown in Figure 4.4. The *Cluster* class has methods which pertain to the details of a cluster. Getters and setters are used to

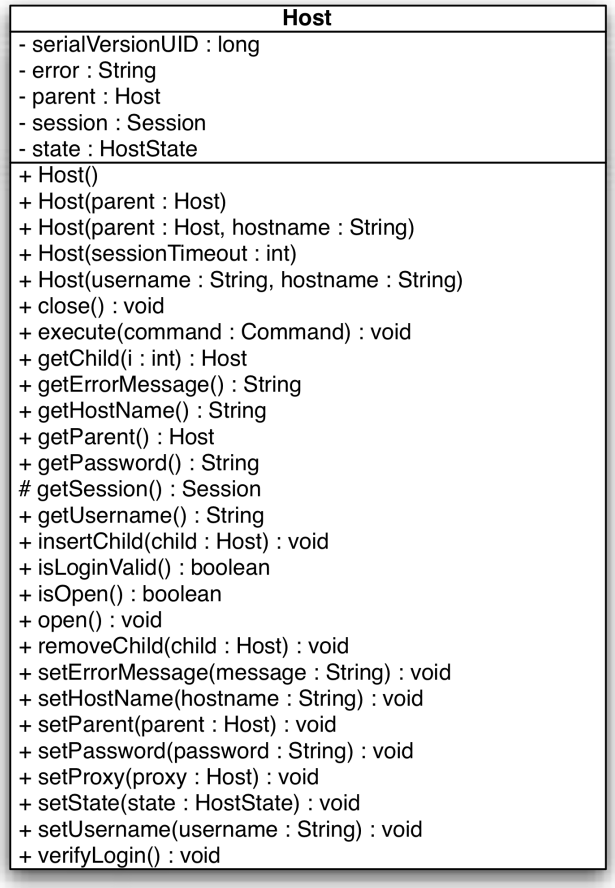

Figure 4.3: *Host* Class Diagram

keep track of the data related to the cluster. Figure 4.5 shows these methods in the class diagram.

4.2.4 Host State Classes

The *State* design pattern [14] is used to implement the state of the *Host* objects. The *HostState* classes are shown in Figure 4.6. The *HostState* class plays the *State* role, while the other classes are *ConcreteStates*. The *Host* plays the *Context* role in this implementation of the *State* pattern.

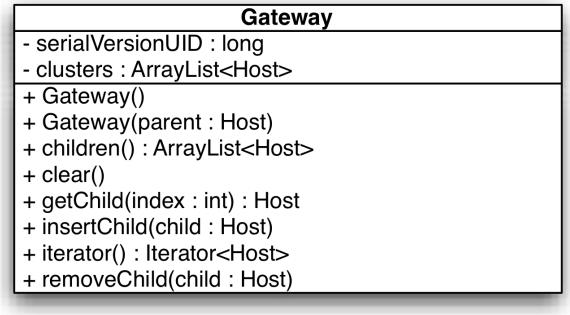

Figure 4.4: *Gateway* Class Diagram

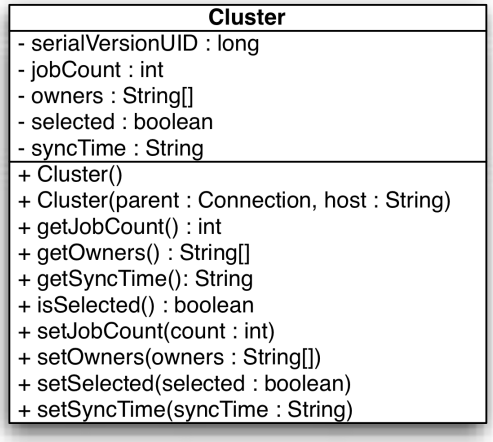

Figure 4.5: *Cluster* Class Diagram

A state diagram for these states is shown in Figure 4.7. Every *Host* begins its life in the *LoginNotVerified* state. Once login information, hostname, username, and password, has been verified, the *Host* will move to the *LoginVerified* state. Following an *open* operation, the *Host* will be in the *OpenState*. The *Host* must be in the *OpenState* in order for a command to be executed. While a command is being executed, the *Host* will be in the *ExecutingState*. Once the command execution has been completed, the *Host* will be back in the *OpenState*. A close operation may be performed to move the *Host* to the *ClosedState*. The *Host* provides an interface

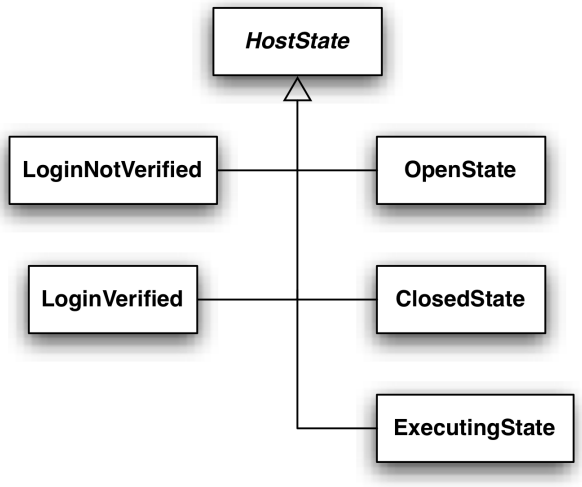

Figure 4.6: *HostState* Class Hierarchy

for clients to make requests. These requests are delegated to the *HostState* object, for the appropriate behavior for each request, depending on the state. A class diagram for *HostState* is shown in Figure 4.8.

4.3 Job Data

4.3.1 SQLite Databases

The job information data must be persistent, due to the nature of Android applications. At any time, without warning, the application can be paused or destroyed by the user or the operating system. Rather than waiting until that happens, the job data will be saved to a file as soon as it is acquired. One benefit of the Android devices over traditional computer desktop systems is that the persistent memory of the system is typically solid state flash memory, instead of mechanical hard disk drives. As a result, the access time is much faster.

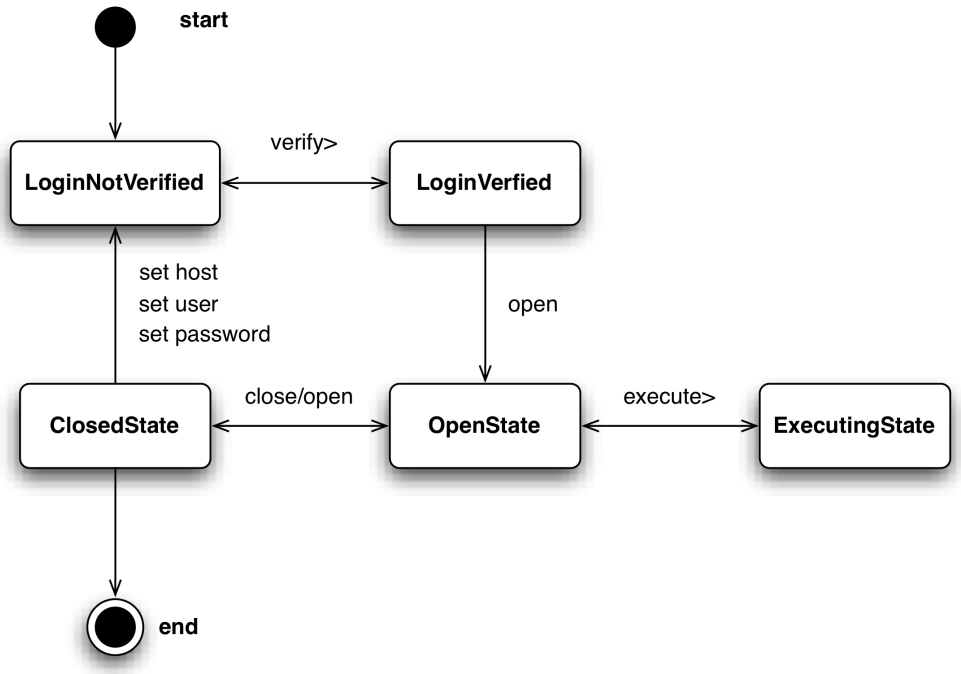

Figure 4.7: *Host* States

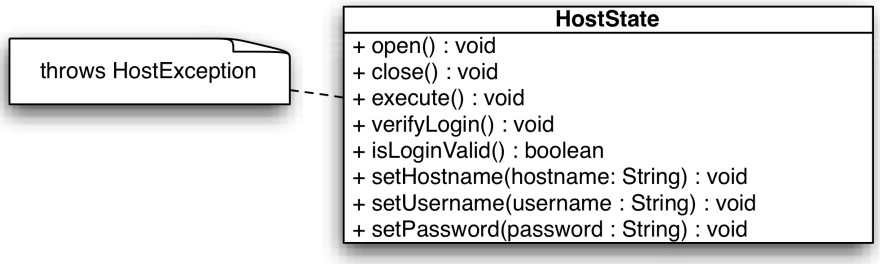

Figure 4.8: *HostState* Class Diagram

Additionally, since a file will be used to persist the data, a SQLite database is used to take advantage of the powerful query operations available. Also, the Android development framework provides classes which make it convenient to handle these databases and display query results.

Initially, the design involved a single database, with a single table for job infor-

mation. That table is shown in Figure 4.9.

jobs
id : int primary key
job_id : int
username : text
job_name : text
session_id : text
node_count : text
status : text
elapsed_time : text
nodes : text
tasks_req : text
memory_req : text
time_req : text
selected : int
begin_tracking : int
stop_tracking : int
elapsed_time_minutes : int

Figure 4.9: Job Table

Since the requirement for multiple clusters was not known until after the design and implementation was well under way, it seemed that the simplest way to support handling multiple clusters connected to a single gateway was to provide a SQLite database for each cluster. As a result, there will be a single database file for each cluster.

This database design should be adequate for supporting thousands of jobs per cluster. The *id* field is the primary key, as is typical in most SQLite database tables. It has a unique integer value. The *selected* field represents a boolean value for whether the user has selected the given job. A zero value represents *false*, while a value of one represents *true*. The field, *begin_tracking*, is a boolean value which signifies that the

job is a newly added job. The field, *stop_tracking*, is a boolean value which is used to determine if a job should be purged from the database, since it is no longer being tracked. A job not being tracked means that it is no longer showing up in the job information data received from the cluster. The last field, *elapsed_time_minutes*, is an integer value representing the number of minutes that has elapsed since the job started.

The remaining fields are fields that directly result from the *qstat* command. Table 4.1 summarizes them.

Table 4.1: Fields Related to the qstat Command Output

Field	**Description**
job_id	job identifier assigned by PBS
username	job owner
job_name	job name given by the submitter
session_id	the session id of a running job
node_count	number of nodes requested by the job
status	the job's current state
elapsed_time	amount of CPU time used by the job
nodes	nodes the job is running on
tasks_req	number of CPUs requested by the job
memory_req	the memory requested by the job
time_req	the CPU time requested by the job

4.3.2 JobsData class

The *JobsData* class provides the interface to the SQLite database for the job information. Although there are object relational mappers (ORMs) available for use with Android [9], this design does not use an ORM. Instead, the SQLite database management classes provided by Android are used.

As shown in Figure 4.10, a *JobsData* object is composed of a *DatabaseHelper* object. *DatabaseHelper* extends the *SQLiteOpenHelper*, which is one of the SQLite

database management classes provided by the Android framework. The *Database-Helper* overrides two methods, *onCreate* and *onUpgrade*. The *onCreate* method determines how the database structure is formed. In this case, it is mainly a CREATE TABLE statement, in SQL syntax, which is executed when the database is created for the first time. The *onUpgrade* method is provided in order to support upgrading the database structure. This may involve adding columns to a table, adding tables, and possibly more.

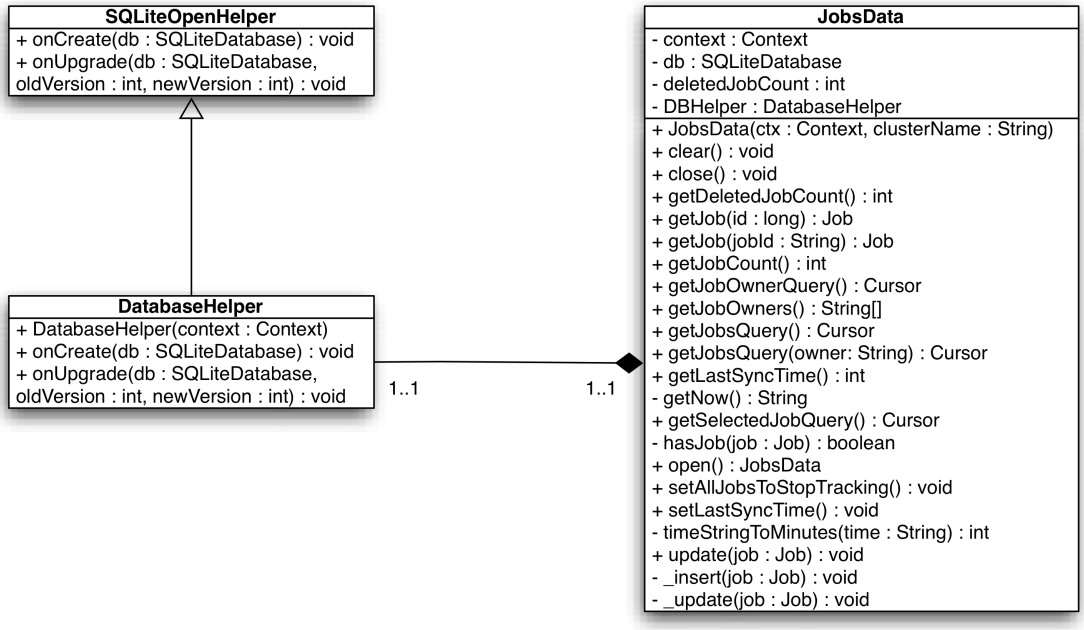

Figure 4.10: *JobsData* class diagram

JobsData provides four methods which return *Cursor* objects. The *Cursor* class comes from the SQLite classes provided by Android and is the result of a database query. The reason these methods provide the cursor object, rather than an array, is that the *Cursor* is used in some of the view classes to display data with the UI. This will be discussed more in later sections. Each *Cursor* object should be closed when

it is no longer needed. This is the responsibility of the calling method.

JobsData also has *open* and *close* methods for opening and closing the database. The database should be closed when it is no longer needed. This may be done when a *Cluster* is deleted from the cluster list, or when the *JobsService* is desroyed. The *JobsService* would be destroyed some time after the last *Activity* is unbound from it.

Setting attributes on *Job* objects does not update the database, as would normally be expected from a ORM scheme. In order to update the database, a *Job* object is passed to the *update* method. If a *Job* exists in the database with a matching job identifier, then the database is updated. Otherwise, the new job is inserted into the database table.

The remaining methods for the *JobsData* class provide the means for getting data that is needed for display. There are two *getJob* methods, which return a *Job* object. The *Job* object can be used to extract job-specific information which can be displayed.

4.3.3 Job Class

A *Job* class is used to manage job-specific information. Figure 4.11 shows the class diagram for *Job*. All of the methods are getters and setters. Garbage collection is an expensive process and should be avoided, if possible. Therefore, extra effort was made to ensure that the number of *Job* objects needed within the application is minimal.

4.4 Android UI

The Android UI component of this application is composed mainly of *Activity* subclasses. Each of the activities represent a single screen which is encountered while using the application. The flow of the activities is shown in Figure 4.12.

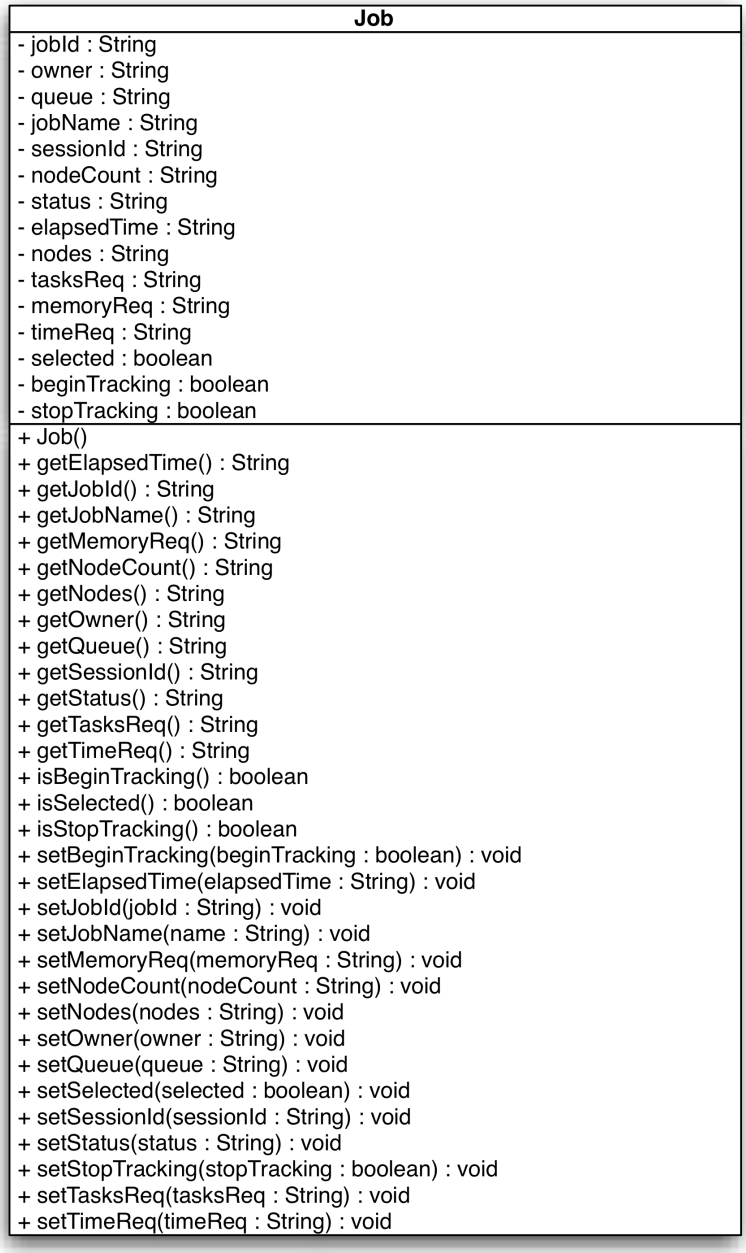

Figure 4.11: *Job* class diagram

4.4.1 ClustersActivity

The application begins by entering the *ClustersActivity*. When the *ClustersActivity* is created, it is bound to a *JobsService* object, which will be discussed in a later

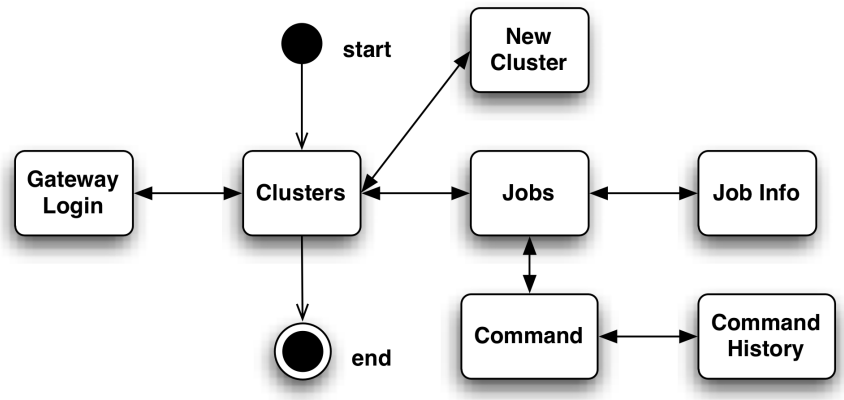

Figure 4.12: Application Activity Flow

section. This screenshot of the *ClustersActivity* is shown in Figure 4.13. The user may log in to the gateway by pressing the *Login to Gateway* button. Once the button is pressed, the *GatewayActivity* is started, where the user can enter the gateway *host name*, *user name*, and *password* fields. Returning to the *ClusterActivity* is achieved by pressing the *Login* button. If the login was successful, the gatway's host name will appear in the button at the top of the screen. Otherwise, an error message displaying the reason for the failed login attempt will be displayed. After a successful login, the application will return to the *ClustersActivity*, and it would appear as in Figure 4.14.

If the login was successful, the other buttons, *add* and *del* will be enabled. Cluster master node names may be added or deleted using these buttons. Pressing the *add* button will start the *NewClusterActivity*.

The clusters displayed by the *ClustersActivity* represent the clusters whose jobs will be monitored. The user may add a cluster to the list, by pressing the *Add* button. The *ClustersActivity* employs a *ListView*, which is provided as part of Android. Usually, *ListView* objects display simple text strings by default, for an item in the list. One way to customize the display of the item, and make it a bit more

Figure 4.13: Initial *ClustersActivity* Screen

Figure 4.14: Return to the *ClustersActivity* After a Successful Login

complicated compared to displaying only a simple string, is to use an *ArrayAdapter*.

4.4.2 ClusterAdapter

ClusterAdapter extends *ArrayAdapter*, which is provided by Android. To customize the view for an item in the list, the *getView* method is usually overriden to provide the custom view. Figure 4.15 shows code that overrides the *getView* method. The layout of the view is provided by a resource defined in an Android XML file.

In addition to displaying relevant data, the *View* which is returned by the *getView* method also has a *CheckBox*, that can be used to select or deselect clusters. Selected clusters can be deleted by pressing the *delete* button on the activity. In order to provide this functionality, a listener is needed to see when the checked state of a *CheckBox* has changed. This is provided by the *onCheckedChangeListener* that is shown in Figure 4.16. The behavior of the listener is simple; it just calls *setSelected* on the *Cluster* and passes in an appropriate value.

4.4.3 Gateway Activity

The *GatewayActivity* class represents the screen which is displayed to allow the user to provide the *host name*, *user name*, and *password* needed to log into the gateway, using an SSH connection. Figure 4.17 shows that the *GatewayActivity* class is a subclass of the *android.app.Activity* class. Three methods are overridden, *onCreate*, *onRestoreInstanceState*, and *onSaveInstanceState*. The last two are overridden to preserve the state when the device orientation is changed. Figure 4.18 shows a screenshot of the *GatewayActivity*.

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
  LinearLayout view;

  Cluster cluster = getItem(position);

  if (convertView == null) {
    view = new LinearLayout(getContext());
    String inflater = Context.LAYOUT_INFLATER_SERVICE;
    LayoutInflater layoutInflater = (LayoutInflater) getContext().getSystemService(inflater);
    layoutInflater.inflate(clusterItemLayout, view, true);
  } else {
    view = (LinearLayout) convertView;
  }
  CheckBox checkBox = (CheckBox) view.findViewById(R.id.cluster_item_checkbox);
  checkBox.setOnCheckedChangeListener(getOnCheckChangeListener(cluster));
  checkBox.setChecked(cluster.isSelected());

  TextView nameView = (TextView) view.findViewById(R.id.cluster_item_name_field);
  TextView jobCountView = (TextView) view.findViewById(R.id.cluster_item_count_field);
  TextView ownersView = (TextView) view.findViewById(R.id.cluster_item_owners_field);
  TextView syncTimeView = (TextView) view.findViewById(R.id.cluster_item_synctime_field);


  nameView.setText(cluster.getHostname());
  jobCountView.setText(Integer.toString(cluster.getJobCount()));

  String ownersList = "";
  for (String owner: cluster.getOwners()) {
    ownersList += owner + ",";
  }
  if (ownersList.length() > 0) {
    ownersList = ownersList.substring(0, ownersList.length() - 1);
  }

  ownersView.setText(ownersList);
  syncTimeView.setText(cluster.getSyncTime());


  return view;
}
```

Figure 4.15: Overriding the *getView* in *ClusterAdapter*

```
private OnCheckedChangeListener getOnCheckChangeListener(final Cluster cluster) {

  OnCheckedChangeListener listener = new OnCheckedChangeListener() {

    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {

      if (isChecked) {
        cluster.setSelected(true);
      } else {
        cluster.setSelected(false);
      }
    }
  };
  return listener;
}
```

Figure 4.16: Providing the *onCheckedChangeListener* in *ClusterAdapter*

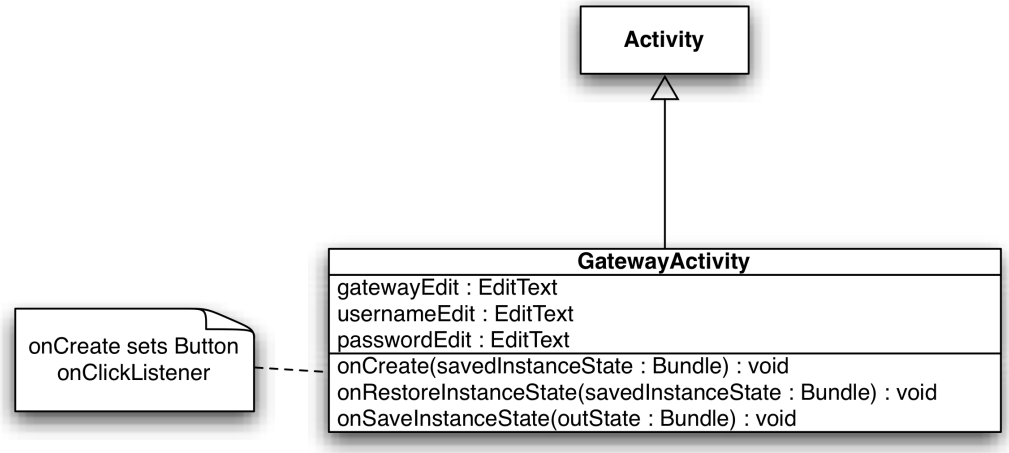

Figure 4.17: *GatewayActivity* Class Diagram

4.4.4 NewClusterActivity

The *NewClusterActivity* class represents the screen where the user can enter information needed to log in to a new cluster. The cluster name, otherwise known as the host name, is required. Also, the user name and password may be provided, or the

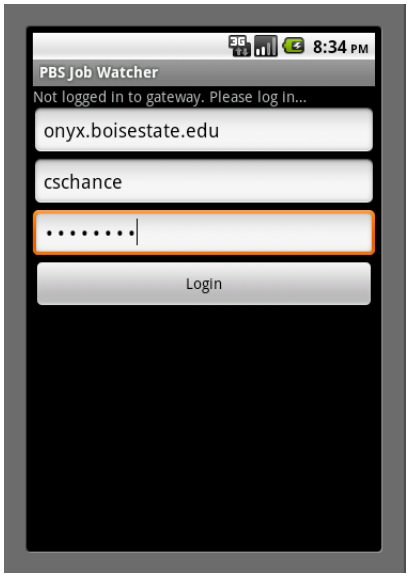

Figure 4.18: *GatewayActivity* Screen

check box may be checked if those values are the same as the gateway. Figure 4.19 shows a screenshot of the *NewClusterActivity*.

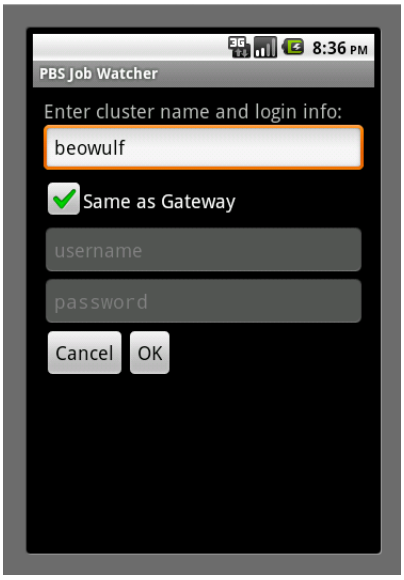

Figure 4.19: Adding a Cluster in *NewClusterActivity*

4.4.5 JobsActivity

JobsActivity displays the jobs in the cluster's PBS job queue. It has a listener, so it can update itself when the job information has been updated by the *JobsService*.

4.4.6 JobOwnersActivity

This activity shows a list of job owners for the given cluster. If a job owner item is clicked, the result is returned to the JobsActivity, where only the jobs for the chosen owner will be displayed.

4.4.7 JobDetailActivity

JobDetailActivity displays all the available information for a particular job. It is composed of several *TextView* objects, which are laid out based on the definition provided by an Android XML layout file. The *JobDetailActivity* object has a listener, so it knows when job information has been updated.

4.4.8 CommandActivity

CommandActivity allows the user to enter a command, as they would normally do with a command line interface (CLI). The *CommandService* is used to actually perform the execution on the particular cluster master node. The result of the command is displayed in the *TextView* in this activity.

4.4.9 CommandHistoryActivity

The *CommandHistoryActivity* simply displays a list of previously entered commands. If one of them is clicked on, then the activity returns the string value to the *CommandActivity* where it is displayed in the text entry box.

4.4.10 JobsPreferenceActivity

The *JobsPreferenceActivity* provides a way for the user to change the shared preferences within the application. The options available in the shared preferences involve the frequency at which the job information is updated by performing communication with each of the cluster master nodes. There are three values to change. The first option is whether to perform the update while the application is running in the background. A *CheckBox* is provided for this. Then next option, the frequency to perform the update when the application is in the background, has a default value of 60 minutes. Finally, the frequency to perform the update when the application is in the foreground, has a default value of three minutes. The layout, default values, and labels for this activity is defined in a Android XML resource file, named *preferences.xml*. This method of implementing a *PreferenceActivity* is typical in Android applications [15].

4.5 Services

The *Services* provide the functionality to perform tasks on the data, which usually involve tasks which take a long time. Sometimes, these tasks may take an indeterminate amount of time, such as network communication. It is good practice to decouple this functionality from the UI, in order to ensure the program is responsive to the user. The following subsections describe the services which are used.

4.5.1 JobsService

The *JobsService* class is the workhorse of the application. Its main responsibility is to keep the job information updated in each of the databases. It does this by setting a

```
private SharedPreferences prefs;
private OnSharedPreferenceChangeListener prefsListener;
...
prefs = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
prefsListener = new OnSharedPreferenceChangeListener() {

  @Override
  public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
  String key) {
    updateFromPreferences();
  }
};
    prefs.registerOnSharedPreferenceChangeListener(prefsListener);
```

Figure 4.20: Setting Up an *OnSharedPreferenceChangeListener*

Timer to perform the next communication with the remote clusters. The time interval to determine when the *Timer* will execute its task is stored in shared preferences. One thing the *JobsService* must do is listen to see if these shared preferences get changed by the user. In the *onCreate* method for *JobsService* the listener is set up. The code snippet in Figure 4.20 shows how this is done. The *updateFromPreferences* method reads the new values from the shared preferences.

Since Android services run on the main UI thread, along with the activities, any long-running task, such as network communcation, must be performed on a background thread. When the *Timer* executes its task, the background thread is started, using the *Runnable* attribute of the *JobsService*, *updateJobs*. This attribute, along with the other attributes and methods of the *JobsService* class is shown in the class diagram in Figure 4.21.

One last function the *JobsSevice* performs is a notification when network communication is underway. The *Notification* classes provided by Android are used. As a result, a notification icon is displayed in the area at the top of the phone's display. If the user drags the bar down to view the notification items, they can click or touch

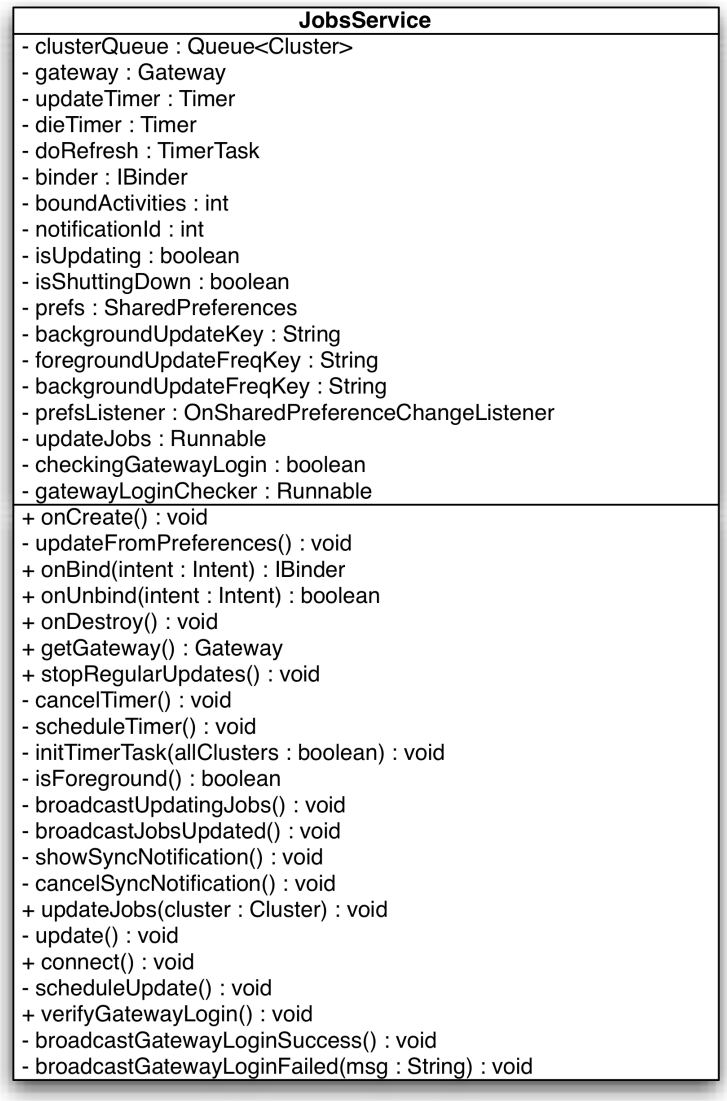

Figure 4.21: *JobsService* Class Diagram

the item for this application. The result will be the *ClustersActivity* is brought to the foreground.

4.5.2 CommandService

The *CommandService* provides the means to execute a command on a given cluster master node. The UI element that interacts with the *CommandService* is the *CommandActivity*. The *CommandService* has a *Gateway* and a *Cluster* objects. Figure 4.22 shows these attributes, along with the other attributes and methods.

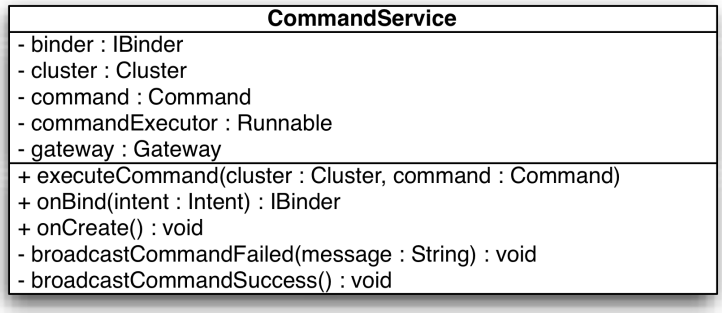

Figure 4.22: *CommandService* Class Diagram

CHAPTER 5

DISTRIBUTION

5.1 Web Site

The application is available for download from the website, `http://cs.boisestate.edu/~amit/research/pbs-app/`. After downloading the .apk file onto the Android device, the application can be installed. The Android device should be running Android version 2.0 or higher.

CHAPTER 6

CONCLUSION

6.1 Requirements

During the development process, the importance of the *requirements* phase was discovered after the design and implementation stage was well under way. Specifically, a potential customer for this application mentioned that their system architecture involved a gateway to the Internet with several clusters connected to the gateway machine. This is a typical configuration, which would probably have been determined during a thorough *requirements* phase. However, as requirements change or features are added, the development process must respond appropriately. This is one of the central ideas behind the agile method of software development [8].

Fortunately, the design was robust enough that it was adapted to the additional requirement. There were a few affected classes, and some classes were added, but the majority of the design was unaffected.

6.2 Testing Challenges

Although unit testing is supported by Android and the ADT within Eclipse, there are some limitations. There is little to no support for object mocking from libraries such as jMock [5]. One of the main benefits for using mocking libraries such as these

is the ability to mock concrete classes [13]. Unfortunately, the Dalvik VM does not support the type of behavior these packages rely on [1].

6.3 Work to be Done

As with any user application, it never truly completed. The following subsections describe the additional work to be done.

6.3.1 Optimizing Network Communication

Depending on the number of jobs in the cluster queue, network communication can take a large amount of time. Some cases took over a minute. There are some things that can be done to optimize network usage. Currently, the application downloads all the job information from the cluster. In some cases, this may not be necessary. For example, if the user is only interested in getting information on jobs for a particular owner, it wouldn't be necessary to update job information for all the jobs, all of the time.

6.3.2 Additional User Features

As the need for additional user features arises, new versions of the application will be released with new features implemented. Some examples of features may include a more extensive set of commands to execute on cluster jobs.

6.3.3 User Interface Enhancements

The entire UI is composed of standard-looking Android components. This is okay for a utility application. However, some improvements could be made, such as more

graphics for buttons, instead of text. Also, different color schemes may improve the look of the application. As new versions are developed, it is likely that the user interface will see some improvements, as well.

6.3.4 Marketing the Application

Although, the Google Play store is one of the most useful tools for marketing an Android application, a web page is also useful. A web page can provide more detailed instructions for using an application, compared to what is usually offered within a mobile application. And, the web page can direct potential customers to the Google Play store, where the application can be purchased. Furthermore, the web page could provide links to related tools.

6.3.5 iOS Version of this Application

Since there is a large user-base of iOS devices, it would make sense to have a version of this application for these devices. I have been exploring how this application would be implemented for iOS.

6.3.6 Optimizing the Application for Larger Screen Sizes

Android tablet devices have been coming to the market within the last year or two. This application does not utilize the large screen real-estate available on these devices. Improvements can be made to optimize the application for tablet devices.

6.4 Experience

Developing an application for a mobile device turned out to be a different software project, of sorts. There is some overhead that needs to be taken care of to manage to possibility that the application could be shut down at any moment. This requires a different mindset when developing the application. For example, in a traditional desktop application, the user usually determines when the application ends, by choosing *exit* from a menu or clicking the x in the corner of the main application window. It is generally up to the user to decide if they want to save anything beforehand. This isn't the normal mode of operation for a mobile application.

I feel the experience gained from this project will help me complete future projects successfully. This will be true for mobile applications and other projects. All phases of the development process were experienced to a certain extent. However, there could have been more emphasis on testing, up front. While developing some portions of the project, this was the case. For example, unit testing was used, in the spirit of test-driven development (TDD), for implementing the *HostState* classes. Other portions of the project didn't receive the same treatment, although it would have been beneficial. And, I would have liked to explore the notion of design-driven testing (DDT), which I have recently heard about.

Developing software is continually a learning process. With the advent of mobile devices, there is some learning curve for developing for these new platforms. As of this writing, mobile devices with multi-core processors are starting to come to market. It will be interesting to see what the future holds for software development for these devices and future ones.

REFERENCES

[1] Android Testing. https://sites.google.com/site/androiddevtesting/.

[2] Eclipse Foundation. http://www.eclipse.org.

[3] Git Hub. http://www.github.com.

[4] Google Play Android App Store. https://play.google.com/store/apps.

[5] jMock - An Expressive Mock Object Library for Java. http://www.jmock.org/.

[6] JSch - Java Secure Channel. http://www.jcraft.com/jsch/.

[7] LIBSSH2 - The SSH Library. http://www.libssh2.org/.

[8] Manifesto for Agile Software Development. http://agilemanifesto.org/.

[9] Ormlite - Lightweight Object Relational Mapping Java Package. http://ormlite.com.

[10] SQLite. http://www.sqlite.org/.

[11] The Secure Shell Connection Protocol. http://www.ietf.org/rfc/rfc4254.txt, 2006.

[12] Oleg Andreev. GitBox. http://www.gitboxapp.com, 2010.

[13] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided By Tests*. Addison-Wesley, 2010.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Indianapolis, IN, 1995.

[15] Reto Meier. *Professional Android 2 Application Development*. Wiley, Indianapolis, IN, 2010.

[16] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice Hall, second edition, 2005.